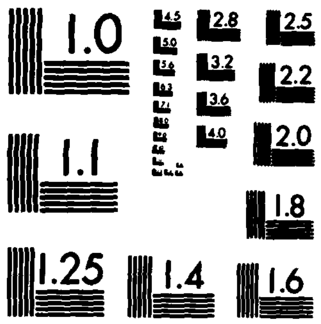
AD-A153 378 SOFTWARE ENVIRONMENT FOR CONDUCTING INTERACTIVE STATISTICAL ANALYSIS(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING 1/2

UNCLASSIFIED M F AMELL 14 DEC 84 AFIT/GCS/MATH/84D-2 F/G 12/1 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

SOFTWARE ENVIRONMENT FOR CONDUCTING INTERACTIVE STATISTICAL ANALYSIS

THESIS

AFIT/GCS/ENG/84D-2

Mark F. Amell
Captain, USAF

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/MATH/84D-2

SOFTWARE ENVIRONMENT FOR CONDUCTING INTERACTIVE STATISTICAL ANALYSIS

THESIS

AFIT/GCS/ENG/84D-2

Mark F. Amell
Captain, USAF

AFIT/GCS/MATH/84D-2

SOFTWARE ENVIRONMENT FOR CONDUCTING INTERACTIVE STATISTICAL ANALYSIS

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Mark F. Amell

Captain, USAF

December 1984

Approved for public release; distribution unlimited.

PREFACE

The purpose of this study was to develop a software environment for conducting statistical analysis of exponential failure models interactively. The environment was successfully completed and thoroughly tested.

The method used in this study to develop the interactive environment can be used to extend the capabilities of this system by adding the ability to process other distributions or unrelated tasks.

In developing this thesis effort I am particularly indebted to my thesis advisor, Dr. Panna Nagarsenker, for her continuing patience and assistance in times of need. I also wish to thank my wife, Nancy, for her understanding and support.

Mark F. Amell

Table of Contents

List of Figures

AFIT/GCS/MATH/84D-2

ABSTRACT

This thesis ~~study~~ developed a software environment for conducting statistical analysis of exponential failure models interactively. Exponential failure models will work for small amounts of data and are particularly valuable when gathering large amounts of data is not practical, such as loss of lives or airplanes.

The design of the system down to the individual modules demonstrated that by using the software engineering techniques explored, an environment can be efficiently created. The approach used in this project not only created a valuable tool, but it's use is encouraged in further developments for other distributions. Additional keywords: computations; programming manuals; test and evaluation; computer program verification.

I. Introduction

The primary task of this thesis effort is to develop a software environment for conducting statistical analysis of exponential failure models interactively.

Background

Failure rates have many applications and uses; for example, 1. the weather radar system on a 747 aircraft has a mean time between failure(MTBF) of 1,140 hours. Assuming an exponential time to failure distribution the following questions can be answered: (1) What is the probability of failure during a four hour flight? (2) What is the maximum length of a flight such that the reliability of the system will not be less than 0.99, assuming the system is in continuous operation during the flight? (3) How often should the weather radar system on a 747 aircraft be completely serviced?

2. A military vehicle has a required 200 kilometer mission reliability of 97.5%. (1) What is the vehicle's required MTBF? (2) How many missions can be run before exceeding a 10% chance for mission failure?

3. Several different military aircraft have equipment failures. The data is gathered giving intervals in operating hours between the successive failures of the equipment for each aircraft considered. This is an example where we have several series rather than one. A point of obvious interest concerns whether the observations differ significantly between different aircraft. This is the point of primary interest in considering, for example, matters of preventive maintenance.

4. Failure times of several electronic computers are recorded and if the Poisson process is observed, then the interesting point to discover is whether all the computers have the same average failure rate.

5. A new design is to be tested and if it is judged to be superior to the old design than the old design will be replaced. From extensive testing over the years the old design is known to have an MTBF of 1,250 hours. If the new design has an MTBF that is 50% better than the old design, we desire an 80% chance of selecting the new design. If the new design is no better than the old design, we desire a 95% chance of not selecting the new design. (1) How many failures will we have to observe? (2) Assuming that the new design has an MTBF of 1,565 hours and all items placed on test are allowed to fail, what is the expected time of

testing? (3) How many items must be tested to obtain a 30% reduction in test time?

While facing s[illegible]h problems a manager wi[illegible]s to have an automated interactive [illegible]tem that helps to answer such questions. The objective of this project is to pr[illegible] such a facility, hence to create a software environment for conducting the statistical analysis described below.

The problems in all the above cases reduce to testing the hypothesis that p samples have been randomly drawn from the same exponential distribution. The hypothesis can be tested by using the likelihood ratio criterion and for $p = 2$ this reduces to an F test. Bartlett's M test for homogenity of variances in samples from a normal population can also be used to examine this hypothesis for a general p.

The hypothesis can be tested by using the likelihood ratio criteria, it's exact distribution and percentage points were obtained by Dr. Nagarsenker in 1980, a review of which follows:

Suppose that p samples of equal size n are available and that the ith sample contains observations x_{ij} with mean $\bar{x}_i$ $(i = 1,\ldots,p;\ j = 1,\ldots,n)$ and has been drawn from an exponential distribution with an exponential distribution of mean θ_i. The likelihood ratio criterion for testing the hypothesis $H_0: \theta_1 = \cdots = \theta_p$ against a general alternative is

$$\lambda = L^n = \left\{ \left(\prod_{i=1}^{p} \bar{x}_i / \bar{x} \right) \right\}^n \tag{1}$$

where $\bar{X}$ is the mean of the combined sample. It can be easily shown that

$$E(L^h) = E(\lambda)^{h/n} = \left\{ p^h \Gamma(n+h) / \Gamma(n) \right\}^p \Gamma(np) / \Gamma(pn+ph) \tag{2}$$

If we use the inverse transform in (2) the density of L is

$$f(\ell) = k(p,n) \frac{1}{2\pi i} \int_{-i\infty}^{i\infty} \ell^{-h-1} p^{ph} \left\{ \Gamma(n+h) \right\}^p / \Gamma(pn+ph)\, dh \tag{3}$$

where $k(p,n) = \{\Gamma(n)\}^{-p} \Gamma(np)$. Putting $n+h=t$ in the integrand on the right-hand side of (3) we have that

$$f(\ell) = k(p,n) \ell^{n-1} p^{-pn} \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} \ell^{-t} \phi(t)\, dt \tag{4}$$

where

$$\phi(t) = p^{pt} \left\{ \Gamma(t) \right\}^p / \Gamma(pt) \tag{5}$$

Using the well-known expansion for $\log \Gamma(x+h)$, we obtain

$$\phi(t) = p^{1/2} (2\pi / t)^v \left\{ 1 + \sum_{r=1}^{\infty} (q_r / t^r) \right\} \tag{6}$$

where $v = \frac{1}{2}(\rho - 1)$ and the coefficients q_r can be successively computed using the following recurrence relations:

$$q_r = r^{-1} \sum_{k=1}^{r} k A_k q_{r-k}, \quad q_0 = 1 \tag{7}$$

where $A_r = (-1)^r (1-\rho^{r+1}) / \{r(r+1)\} B_{r+1}(0)$. Since $\phi(t) = O(t^{-v})$, we can expand it as a factorial series in the from

$$\phi(t) = (2\pi)^v \rho^{1/2} \left\{ \sum_{r=0}^{\infty} R_r \Gamma(t) / \Gamma(t+v+r) \right\} \tag{8}$$

where the coefficients R_r can be determined using the following relations:

$$\sum_{j=0}^{r} R_{r-j} C_{r-j,j} = q_r, \; R_0 = 1, \; C_{r,0} = 1, \; C_{r,j} = r^{-1} \sum_{k=1}^{r} k A_{kj} C_{r-k,j} \tag{9}$$

where

$$A_{rj} = (-1)^{j-1} \{ B_{j+1}(0) - B_{j+1}(v+r) \} / \{ j(j+1) \} \tag{10}$$

Using term by term integration, we have the exact distribution of L in the form

$$Pr(L \le x) = (2\pi)^v \rho^{1/2 - \rho n} \frac{\Gamma(n\rho)}{\{\Gamma(n)\}^{\rho}} \sum_{r=0}^{\infty} \{ D_r I_x(n, v+r) \} \tag{11}$$

where $D_r = R_r / \Gamma(n+v+r)$, $v = \frac{1}{2}(\rho - 1)$ and $I_x(a,b)$ is the incomplete beta function.

Problem Overview

The system created for this software environment consists of these subsystems described as follows. Subsystem one is to calculate the likelihood ratio l-value from the data points given by the user. Subsystem two calculates the theoretical value. Subsystem three will compare the results from subsystems one and two and report it's result.

The user will interact with the system by answering questions posed by the computer. At the start of the system the number of samples and the sample size will be asked for. The user can then test for multiple alpha levels of those samples. All the user's answers will be first tested to see if they are meaningful; then the data will be processed.

The data that is to be used for statical analysis may be read from a data file or obtained interactively from the user. One of the interactive questions is the name of the file where the data is stored. Other specifics of how to run the system are contained in the user manual in Appendix C.

The problem was attacked by breaking down the entire system into several subsystems. Each subsystem was complied separately using the facilities of the UNIX operating system. The following paragraphs give an overview of UNIX.

Unix Operating System

The following paragraphs describe some of the advantages and disadvantages of using the UNIX operating system to implement and use a program as developed here. Even though the computer and operating system to be used in this project was not decided by this study, the study did provide a better understanding of how to exploit UNIX's best features.

UNIX was written by Ken Thompson and Dennis Ritchie for their own use in studying operating systems and other programming projects. They succeeded in creating a system which would allow them to easily develop programs. UNIX was developed to work interactively instead of in the more common batch approach. A good text processing facility and text editor were added because they were found to be crucial to high productivity of programmers.

A flexible file system contributes directly to increasing a programmer's capabilities. (1; 16; 21; 22) The directory system is stored as a tree structure and allows access to files by following paths. File protection is very versatile and can be set at various levels from a branch of the index to individual files. A set of protection flags includes read, write, and execute flags for the owner, group members, and the rest of the users.

The operating system is written as C modules which form the C shell. Commands to the system are given by executing one or more of these system modules. Having direct control over the operating system gives the programmer access to many powerful tools. One tool that was used heavily in this thesis effort was Makefile. Makefile allows modules to be compiled independently and then linked together to form a system. In addition, Makefile only recompiles modules that have been changed since they were last compiled, therefore saving unneccessary processing time.

Although UNIX does accomplish its main goal of allowing the programmer to easily write and debug programs, the user interface is not very friendly. The CAT command, short for concatenation, is used to list the contents of a file on the terminal. The ls command, which is used to list directories of files at the terminal, has 19 different switches that can be used separately or in combination. These shortcommings though, which just take time to learn, are more difficult for the occasional user than the more frequent user. (15)

The large number of installations using UNIX also makes using it desirable because systems created on UNIX are easily transported to other computers running UNIX. This high portability makes systems even more valuable because of the large number of users that can access the system.

Software Engineering Concepts

Before the principals of structured programming, modularization, and hierarchical structure are used to develop the the design of the software environment, these terms and their concepts will be discussed. The references given are for those interested in further study of these software design methods.

Structured Programming

"Structured programming", perhaps the most abused term in the modern computing literature, derives from the work of Dahl, Dijkstra and Hoare (3). Due in part to some unfortunate remarks in Dijkstra's paper (5), some people came to believe that structured programming was any one of goto-less programming, stepwise refinement, topdown design, or programming in ALGOL-like languages. The essence of structured programming, which is expounded below, was temporarily lost to the literature.

The profound aspects of structured programming concern the use of techniques for reducing the complexity of the programming problem. By emphasizing the "structure" of algorithms, programming sections, or data structures, it becomes possible to separate the behavior of the program at one level from the details of each of the components. Hence, for example, it is useful to refine a program in steps because the skeleton of the program can be shown to behave correctly given some properties of the (unexpanded) subprograms. Each of the subprograms can then be in turn, in isolation from each other and from the program skeleton in which they are embedded.

Stepwise refinement is an example of a structured programming technique. It is a tool by which a programmer can record one aspect of the complexity of a program in order to direct attention to another aspect. The process of organizing the complexity of the problem is accomplished by the programmer, not the technique! Other important techniques in structured programming include the definition of abstract data types (25; 8; 11), hierarchical ordering of program segments, and use of verifiable control structures and operators.

Modularization

Parnas consolidated the informal concepts of modularization (18). Other concepts, such as seperate complilation, are not now included under this term. The primary attribute of a module is that it hides a unit of information. This may be a small implementation detail, such as the actual location of a logical device, or a major design decision, as in whether to sort a KWIC index before printing it or at the same time(19).

Modules contain and hide information. Sometimes they can be encoded as programs or program segments, but not always. For example, the method for ensuring data integrity over procedure calls (the "calling sequence") can be contained in a module, but the actual method may not be the same for two different caller/callee pairs. The syntax of a command language, for example, is independent of the means by which it is represented in a program.

The recent exploration of abstract data types is an important contribution to the process of modularization. For many purposes, a module can take the form of a data representation combined with a set of permissible operations on that representation. Properly described, it serves as an implementation of a mathematical object that can be discussed seperate from its implementation. This permits the abstract behavior to be made available to a program that used the data type while hiding the mechanisms by which behaviors are accomplished. This form of modularization is so widely applicable that it has become the primary emphasis in the design of several new programming languages (2; 7; 9; 12; 24; 25).

The process of modularization results in fragmentation of the system representation. When systems were monolithic expanses of source code, recompiling the system was all that was needed to integrate the system into runable form. The problem of systematically combining modules into systems is basically unsolved, although the Mesa designers have had some success (9).

Hierarchical Structure

The division of systems into levels is most often attributed to Dijkstra (6). Most of the experience in this field has been provided by operating system projects (10; 13; 14; 23). Parnas (17) and others have argued for such organization of other systems.

A hierarchically organized system has an enforced partial ordering on some objects in the system. Such a system may be described as a sequence of levels, each level being defined as the class of objects that use inferior (according to the relation) to objects in higher levels and superior to those in the lower levels.

Careful selection of objects and relations can result in a system with some very nice properties. By using "processes" as the objects, and "provides work to" as the relation, freedom from deadlock in the system follows immediately (6). If "functions" are restricted by "calls" relations, so that function calls always decend a level in the system, stack depth can be limited to a known maximum. And, if sets of functions are ordered by "functional dependency" (10), each layer corresponds to a virtual machine that can be programmed without dependence on the upper levels.

Hierarchy in a system is often established within the information hiding lines. Although this has the advantage of simplicity, difficulty arises in real systems either with establishing a hierarchy at all, or in efficiently implementing the system along the hierarchy established. Since the two concepts are not dependent, hierarchy can be established "orthogonally" to modularization, and the benefits of both structures will obtain in the resulting system (10).

The clear specification of the hierarchical relation is necessary to a clear understanding of the system structure that will result (20).

Design

Each system was developed using top-down design methodology. The basic principle in top-down design is to work from an abstract functional description of a problem (top) to a detailed solution (bottom). (4:A50)

The separately developed modules increase the reliability, maintainability, and adaptability of the entire system. The system is reliable because each module's results can be easily tested against hand calculated results. Maintainability is increased by the modularity of the system because changes or improvements to the system can be easily changed and reverified within the module. Modularity also increases adaptability of the system, a feature which is important for future maintenance and additional applications.

Each subsystem will be further designed using top-down methodology. The subsystems will be implemented on a Vax 11/780 using the tools of the UNIX operating systems to facilitate the development of entire system.

Sequence of Presentation

The main theme thoughout this thesis is to develop a system which provides a comfortable environment for the above described statistical analysis. Chapter 2, entitled "Project Design" describes the subsystems which make up the environment being created in this thesis effort. Chapter 3 describes the algorithms in detail for each of the modules which make up each system. Chapter 4 illustrates the methods used to test the system and summarizes the test results to demonstrate the accuracy of the test data. Finally, chapter 5 summarizes the results and makes suggestions for further work in this area.

Appendix A contains all the source listings. Appendix B contains detailed test results, and Appendix C is a users' manual for the system created.

II. Project Design

Top-Level

The first step in the design process is to interpret the overall requirement, described in chapter one, into a series of steps, each step being defined by one functional statement. Each step is then further refined until the steps at the lowest levels can be easily programmed into a self-contained group of computer instructions often referred to as a procedure or function. The decision as to when to stop refining is usually a matter of judgement by the designer, but some of the factors to consider are the reusability, complexity, and length of the modules being created.

The overall project requirements are as follows. Given the number of exponential populations, size of the samples, and the sample values, the overall system must perform the statistical analysis necessary to test the hypothesis that the given populations are identical.

As analyzed in the first chapter the overall system is divided into three subsystems. The requirements for system one is to calculate the liklehood ratio l-value from the sample values provided by the user.

System two is required to calculate the theoretical percentage points needed for the analysis. System three's requirements are defined as performing the final analysis by using the results from systems one and two. Finally, a driver module is designed which would input the needed responses from the user, execute systems one, two, and three and report the final result to the user.

System one is contained in the procedure lmodule. The only procedure lmodule needs to call is getreal. The general purpose procedure getreal acts as an interface between the computer program and the user. getreal allows the user to enter the necessary data for the program.

The main module thesis would input the needed responses from the user, execute systems one and two, and finally output the results.

System two will use Newton's iterative method to calculate the theoretical value. Module newton requires R coefficients. The R coefficients are obtained by calling procedure modR which does the actual calculations. Finally newton needs to call a probability function called prob. The reason for introducing the modules modR and prob this early is discussed in the following paragraph.

The module modR is called from the main module thesis because this allows many calls to the module newton with different alpha levels without recalculating the R coefficients. The probability function chosen does not converge making the number of terms needed to be infinite, although the function does come close to convergence after only a relatively few terms. The main module determines the number of terms needed by calling the probability functions a number of times and summing the results until the sum is close enough to one.

The user inputs needed for the overall project are the number of samples taken, the size of each sample, if the data is stored in a text file the name of the file, and the alpha level to be tested for. The alpha level is one minus the probability that the final result given by the program is accurate. The size of each sample must be the same, therefore the sample size is asked for only once. If the number of samples and sample size has not changed the user is allowed to test for different alpha levels without waiting for the re-execution of system one and parts of system two that do not change, for example the R coefficients calculated in the module modR are independent of the alpha level being tested for. The general purpose routine getreal can be used to input any numbers.

System three compares the results from systems one and two and reports the results. System three must call procedure lmodule, which represents system one, and procedure newton, which is the interface to system two. The driver module and system three are both contained in the main procedure thesis.

Before the design becomes too detailed an understanding of the people who are going to use the system is needed. The assumption made for this project is that the users understand the statistics involved but are not necessarily familiar with this project or computers. The process of designing a program that is easy to use for an expected group of users comes under the general term of "user friendly".

In the above discussion of the top level the rather general requirements given in chapter one have been transformed into logic of the thesis module. The thesis module will be actually coded after chapter three's detailed design, but the necessary modules are beginning to be defined and are further illustrated in Figure 1.

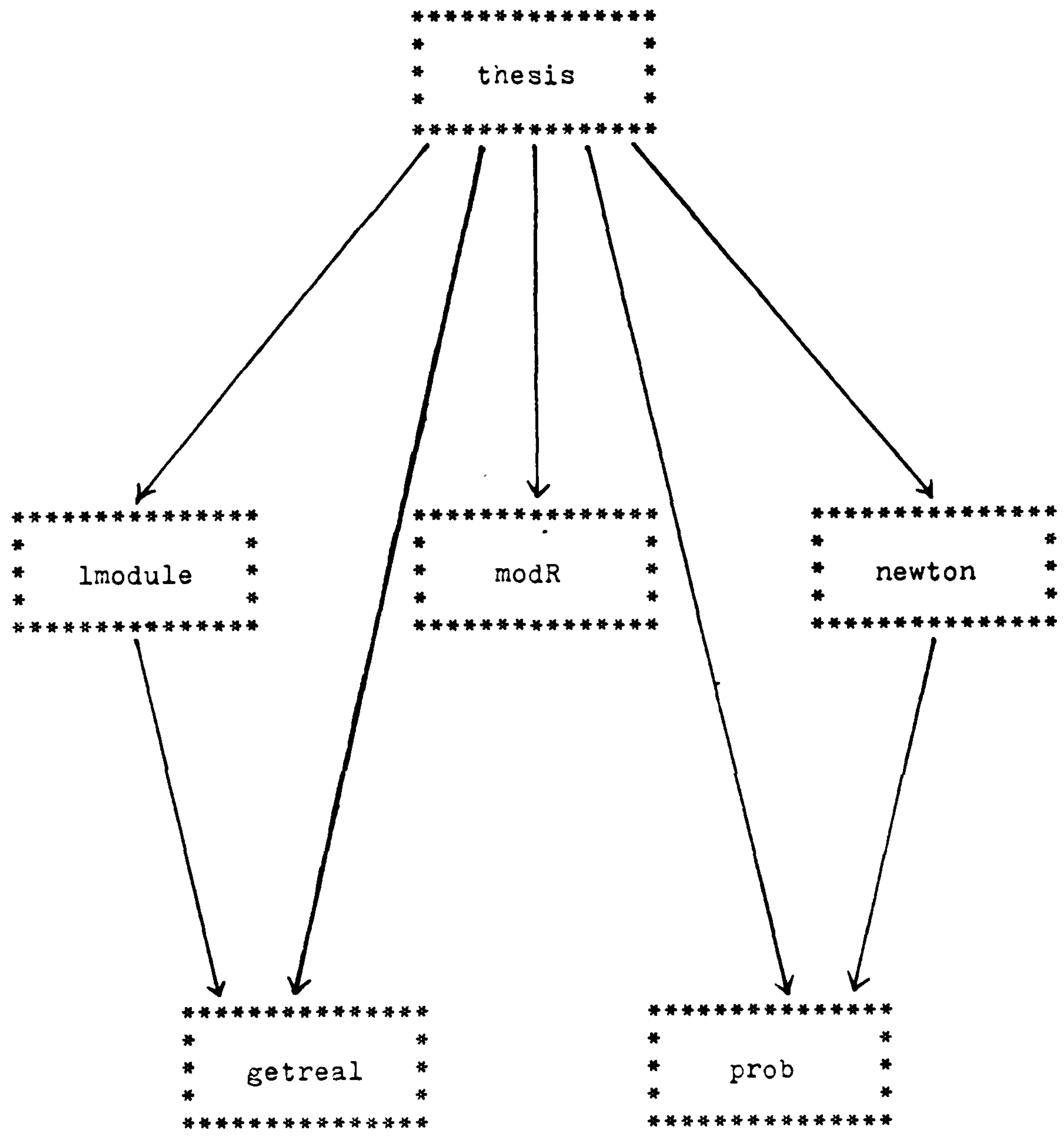


Figure 1 Top-Level

System One

The main module thesis calls system one and passes all necessary information to it. System one, which calculates the l value, only needs to know the name of the file where the inputed data is stored, how many samples, and the size of each sample. The only module system one needs to call is the module getreal to read in the data one number at a time. The l value is calculated as described above by taking the product of the sample means and dividing by the grand mean raised to the nth power where n is the number of samples.

System Two

System two, which calculates the theoretical value, is far more complicated and requires many more modules. The first breakdown of system two is into three steps. Step one is to calculate the R coefficients by calling modR and specifing the number of coefficients required. By knowing the number of samples and the sample size of each, the main module thesis can estimate a maximum number of coefficients required from modR.

The next step is to determine the number of terms necessary to have the probability returned by module prob equal to one when the variable x is set to one. This is done because the function is defined to have an infinite sum of terms, but the function is known to converge close to one after only relatively few terms. The probability is calculated to seven places of accuracy since, after considering the accuracy of system one's caculation of the l value and the precision of the computer, seven places of precision was determined to be sufficient to calculate the number of terms needed by the module newton.

The last step of system two is to calculate a theoretical value to compare system one's l value with by using newton's iterative method described in chapter three's discussion of module newton. The module newton returns the theoretical result system two was created for.

To calculate the R coefficients modR calls modules Queue and createCjr for the Q and Cjr coefficients respectivly. Queue in turn calls ArCoef to calculate the Ar coefficients needed by module Queue. Module createCjr calls createAjr for the Ajr coefficients it needs.

Both modules ArCoef and createAjr call module bernpoly to calculate a value for a specific x calculated by a specific bernoulli polynomial. Module bernpoly calls modules b and com. Module b calculates bernoulli numbers and com

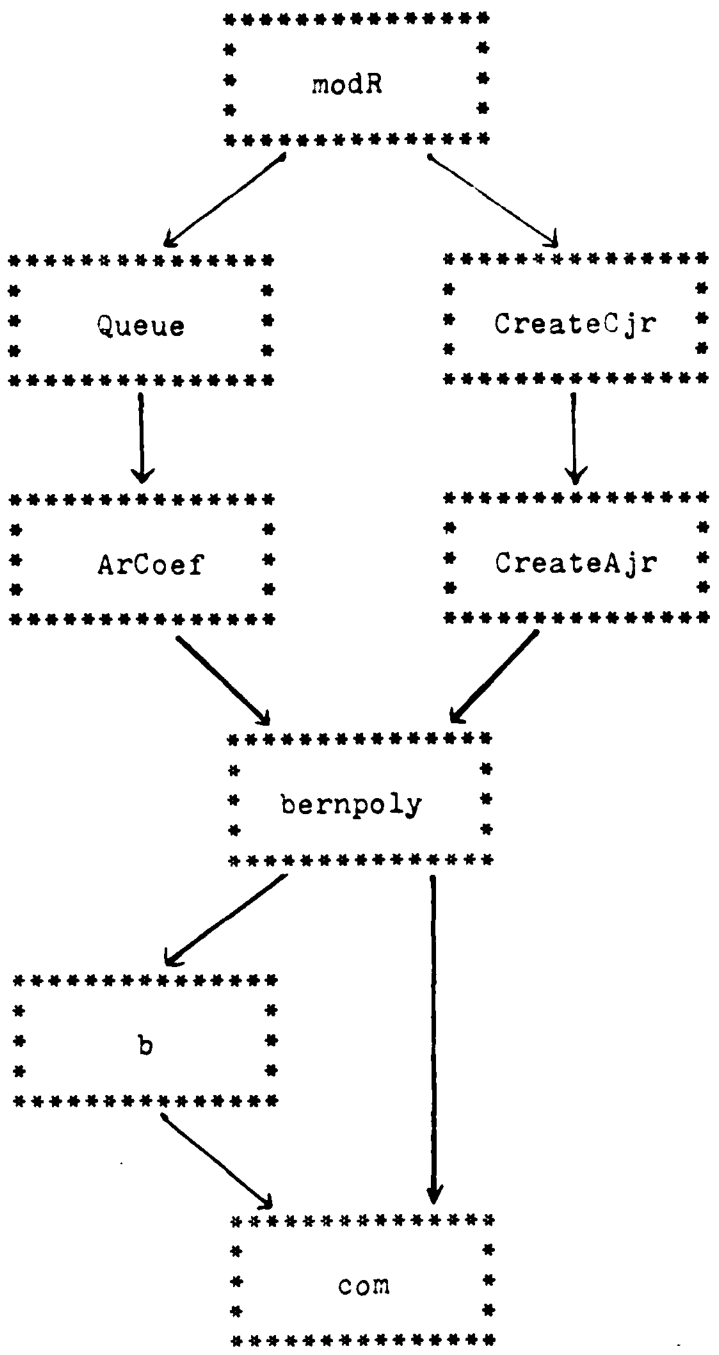


Figure 2 modR

calculates the number of possible combinations as described in elementary probability. Module b also calls module com. The breakdown of modR is further illustrated in Figure 2.

The probability function prob needs to call functions loggam and ibeta. Function ibeta calls function beta, which in turn calls function loggam.

The module newton uses Newton's iterative method to calculate the theoretical value for comparision with system one's result. Module betap calculates a starting value for newton and module prob is called until the theoretical value is obtained. Module prob is described in the previous paragraph and module betap only needs to call function ibeta.

The numerical results obtained by the two subsystems are compared with each other at a preassigned significance level and the result of this comparision is made available to the user. The message will either be: (1) there is sufficient evidence that the data does not come from the same population or that (2) all the data does come from the same population. The breakdown of system two is illustrated in Figure 3.

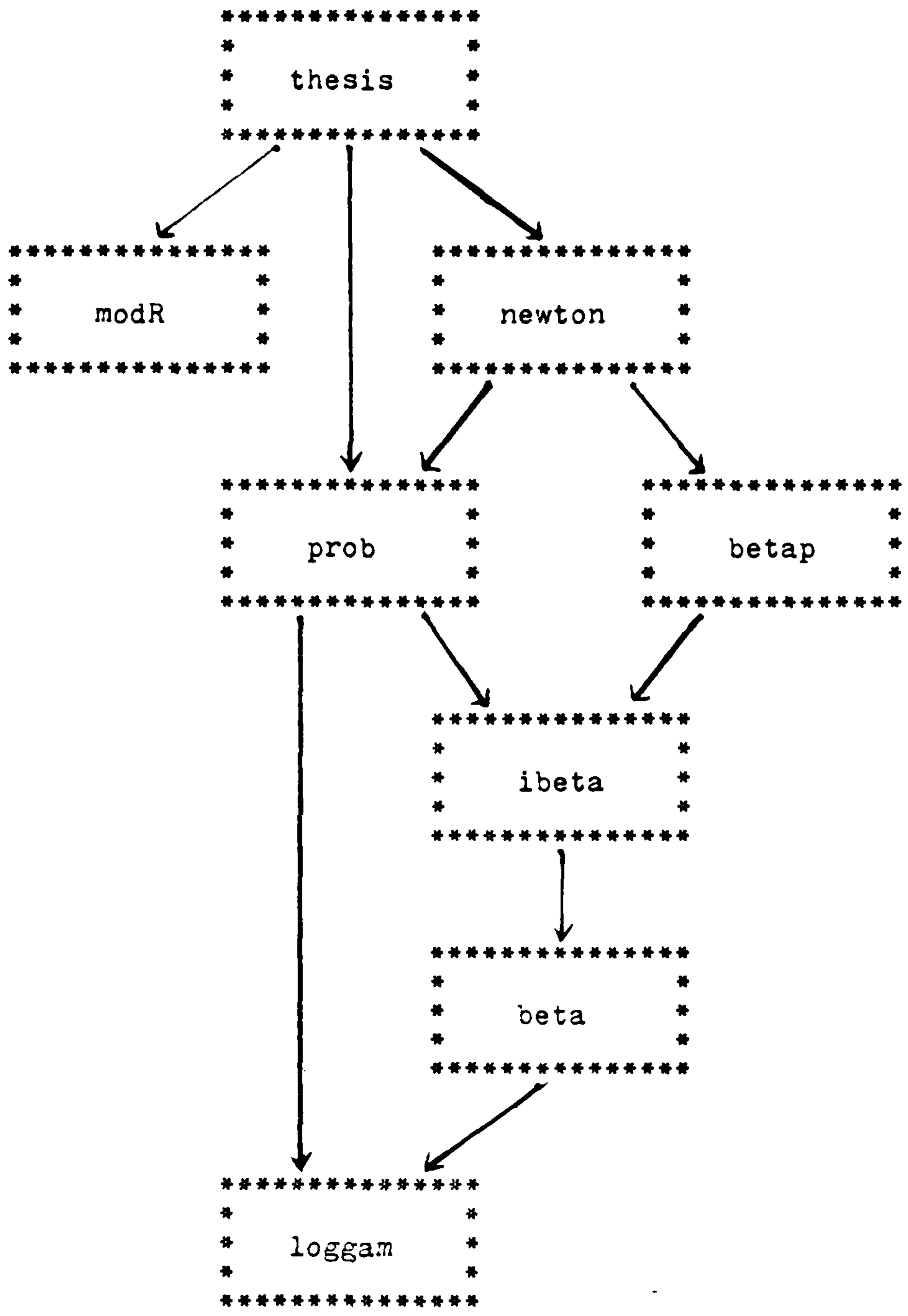


Figure 3 System 2

System Three

System three, as discussed earlier, is contained in the main module thesis. The module thesis, along with all the other modules, will be discussed in detail in chapter three.

As can be seen from this example a complicated project can be easily broken down to single function modules. The modules can then be individually designed and coded. The design of the individual modules is described in detail in chapter three. An additional side affect of modular programming is that many modules can be reused, for example getreal, in the same project or future projects.

III. Detailed Design

General Guidelines

The next few paragraphs explain some of the design characteristics common to all of the modules. Modules, routines, and procedures all refer to a self contained sequence of related computer statements which perform a single task.

Although written to run on a Vax 11-780 computer with a Unix operating system, thesis could run on most systems with few, if any, modifications. These routines should also work with most Pascal compilers due to standardization of the basic keywords used in these procedures.

Global arrays were used to store most of the coefficients calculated in system two. In general the use of global variables is discouraged because of the difficulty in debugging; but, in this particular set of modules the use of global arrays were found to be necessary. Global arrays were used in this set of programs because the same coefficients, at the lower levels, would have to be recalculated thousands of times which would result in a very slow overall response time. For example, the user may have to wait two hours for a response that now takes less than five minutes using global arrays.

Getreal

Getreal is a general purpose Pascal procedure which transfers real numbers from a text file to a Pascal program. Anyone with a need to process real numbers in Pascal could use this routine. Getreal requires two parameters passed to it when called by a program.

NUM contains the real number read in by Getreal and the boolean variable FLAG reflects the status of that read. If the number read in contains a character, Getreal sets FLAG to TRUE to indicate to the calling program that Getreal found bad data. Getreal reads the number in three sequential steps. Deleting the leading blanks and determining the sign of the number precedes the last two steps of reading the integer and decimal parts of the number. A blank or end-of-line character indicates the end of the number. Before discussing these three steps in detail, the following paragraph describes the internal variables required by Getreal.

As previously mentioned, in addition to the variables passed to and returned from the procedure, Getreal requires the internal variables SIGN, CH, I, and DECIMAL. The integer variable SIGN evaluates to either a plus or minus one depending on the sign of the number read in. While Getreal reads the number, CH temporarily holds each digit of the number in character form. The real variable I contains the degree of precision of the most significant decimal place as Getreal reads the number. The boolean flag DECIMAL tells the procedure which side of the decimal point the digits come from.

This first step deletes any leading blanks and sets the variable SIGN to the sign of the number returned. If this step does not find a minus or plus sign, a digit, or a decimal point, FLAG returns to the calling program as TRUE to indicate an error. SIGN is initialized to positive one. Finding a digit or decimal point first implies a positive number and SIGN remains equal to positive one. If an end-of-line condition exists from a previous call, Getreal executes a READLN (read line command) to skip to the beginning of the next line of input data. A WHILE loop (a programming loop that continues to re-execte a group of statements while a condition is true) then reads blanks one at a time until it encounters a non-blank character. Depending on the non-blank character read, Getreal either sets NUM to the digit, DECIMAL to TRUE for a decimal point, SIGN to the appropriate value, or if none of the above, FLAG to TRUE to indicate an error condition.

The integer part of the procedure uses a WHILE loop to read in one digit at a time until it reaches an end-of-current line, a blank, bad data, or a decimal point. Getreal logically executes the integer step between the sign determination and decimal steps as they appear in the real number itself. Getreal converts the characters read to digits by subtracting the ORD('0') (ordinal value of the character zero) from the ORD(CH). Because the computer stores digits in patterns of zeros and ones (ordinal values) in numeric order, the offset from zero gives the value of the digit. Getreal then adds the digit to ten times the previous partial number. As an example of the basic process consider why 5319 equals ((((5*10)+3)*10)+1)*10+9. After finding a decimal point in the imput stream of characters, Getreal starts the final step of reading the decimal part.

In this final step Getreal reads in the decimal part of the real number. A WHILE loop reads in the digits one at a time until it encounters a non-digit character. Converting the characters read in to their numeric form occurs in the same way as described in the integer part. The variable I holds the current precision of the last digit read in. Appending the digits to NUM in this section takes place by dividing the digit by I and adding the quotient to NUM. As an example, consider why .739 equals (7/10) + (3/100) + (9/1000). The final statement in Getreal multiplies NUM by SIGN in order to return a negative number, if appropriate.

Getreal is an efficient general purpose routine to read a real number from a text file into a Pascal program. Pascal has limited input and output capabilities, and therefore relies on user-written routines. Pascal implements faster and more efficient assembler macros; however, importance of error detection often supercedes this advantage in speed. If Getreal detects an error it simply gives a warning and skips the rest of the current line of data instead of abruptly ending the program. Although many alternatives exist, the one presented compares favorably in efficiency and ease-of-use to most others. The steps of Getreal correspond to the format of the real number read. Getreal will also work for integers, and the real fractions inputed can be of the from 0.05 or .05.

lmodule

The function lmodule takes in the input data and calculates the l value for output. The l Value is calculated by taking the product of the sample mean and dividing by the grand mean raised to the nth power where n is the number of samples in the inputed data. The equation is shown below

$$\prod_{i=1}^{n} S_i / G^n \qquad (12)$$

where

n = number of samples

S = individual sample means

G = grand mean of all samples

If running the program interactively the user will be prompted by the count of numbers in the sample, and the user will be able to reenter the number if bad data is entered mistakenly. The batch version will end with a bad data message if bad data is in the file. Both versions except positive real numbers.

The parameters inputed into lmodule are filename, s, and n. filename contains the filename of where the input data is stored or the string "input" to indicate that the data will be entered interactivly. s is the sample size and n is the number of samples. The number of data points needed can be determined by multiplying s times n.

The function lmodule outputs the l value which is later compared to the theoretical value returned by the function newton.

The first step of lmodule is to determine the sample means of each of the samples and multiply them all together. The next step, which is performed at the same time, is to calculate the grand mean. Finally, the l value is calculated by dividing the product of the sample means divided by the grand mean as shown in equation (12) above.

com

The com module is a function which returns the number of combinations obtained from n objects taken r at a time. This result is also known as the binomial coefficient. The binomial coefficients, used by the function bernpoly, are calculated by the equation

$$\binom{n}{r} = \frac{n!}{r!(n-r)!} \tag{13}$$

where

$r \leq n$

$n! = 1*2*3*...*n$

$0! = 1$

Once calculated, the result is returned to the function bernpoly or procedure b for it's use.

The parameters imputed to the function com are integers n and r. Ther are no parameters outputed from module com, but the function does return the real result described by equation (13) shown above.

The module com doesn't need to call any other modules, all the steps required to obtain the result are contained within the module.

Two temporary internal varibales are used by com. The integer j is used as a loop counter and the real variable prod is used to hold intermediate results.

The structure of the module com is two loops. The first loop multiplies prod, which holds the intermediate results, times the quotient of n divided by r. Each time through the loop n and r are both reduced by one. The first loop ends when r is equal to zero. The second loop is similar to the first except that instead of r the internal variable j, which had been initialized to n minus r, is used. Finally, the function returns the value of prod.

There are more efficient ways to program the way combinations are calculated, but the way com was programed is not much slower and does avoid overflow problems associated with other methods. Combinations and binomial coefficients are basic to elementary probability which increases this modules usefulness outside this thesis effort.

b

The b module is a procedure which calculates the bernoulli numbers that are used by the bernpoly module. The bernoulli numbers are calculated by the equation

$$b_n = \frac{-1}{n+1} \sum_{j=0}^{n-1} \binom{n+1}{j} b_j \,,\; b_0 = 1 \qquad (14)$$

where

b () = bernoulli numbers being calculated

com () = calls to the com module

Once calculated, the coefficients are stored in the global array b for the module bernpoly to use.

The only parameter inputed into module b is n, the number of bernoulli numbers required by bernpoly. There are no parameters outputed from module b, all the results are stored in global array b.

Module b does not need to call any other modules. All the required calculations needed to create bernoulli numbers are contained within the procedure.

The temporary internal variables used are j,i, and sum. The integers j and i are sued as loop counters. The outside loop, which ranges form one to n, is controlled by j. The inside loop, which which ranges from zero to j minus one, is controlled by i. The real variable sum holds the summation of the intermediate results of the inner loop and as shown in equation (14) above.

The first step of module b is to set the global variable b flag to true. b flag is set to indicate that module b has already been called once and doesn't need to the called again for the rest of the program. Bernoulli numbers are constants and their recalculation would needlessly slow the overall program down. The next operation is to initialize b[0] to one. The rest of the module consists of two loops, one inside the other. The outside loop counter j represents the current bernoulli number being calculated. The inside loop sums all the terms required to calculate each bernoulli number.

b is an efficient procedure to calculate bernoulli numbers required by the function bernpoly. The resulting numbers are stored in the global array b.

bernpoly

The bernpoly module is a function that solves a bernoulli polynomial for a specific value given the number of terms. This function is used by both modules ArCoef and CreateAjr in their calculations. The general equation is

$$Bernpoly_n(x) = \sum_{k=0}^{n} \binom{n}{k} b_k x^{n-k} \qquad (15)$$

where

n = the number of terms

k = the kth term

x = the value solved for

b = the bernoulli numbers calculated in module b.

Once the equation is evaluated for a specific value of x a single value is returned to ArCoef or CreateAjr.

The parameters inputed into bernpoly are n and x. Both n and x are described aboved.

There are no parameters outputed from bernpoly, only the single value calculated in equation (15).

bernpoly needs to call modules com and b. Module com evaluates combinations as described earlier in this chapter and in elementary probability. The bernoulli numbers are calculated in module b, and stor[illegible] in global array b, only once during the running of the program. The global binary flag b[illegible] is set to true after the first time bernpoly calls module b. Bernoulli numbers don't depend on any variables and the only information module b needs to know is how many numbers to calculate, which is determined in the main module thesis.

The temporary internal variables used by module bernpoly are k, sum, and power. k represents the current term being evaluated, sum holds the summation of the terms previously calculated, and power holds the different powers of x needed in equation (15).

The first step in module bernpoly in to call module b if it wasn't called previously. The next step is to initialize variables sum and power to 0.0 and 1.0 respectivly. The last step is to loop through all n terms, evaluating each as described in equation (15). By summing the terms while the program loops the variable sum will contain the result when the loop finishes.

bernpoly is an efficient procedure to evaluate a bernoulli polynomial for a specific value.

ArCoef

The ArCoef module is a procedure which calculates the Ar coefficients that are used by the Queue module. The Ar coefficients are calculated by the equation

$$A_r = \frac{(-1)^r}{r(r+1)}\left[\frac{b_{r+1}(\rho l) - \rho^{r+1} B_{r+1}(l)}{\rho^r}\right], A_0 = 1 \qquad (16)$$

where

k = the kth coefficient

p = number of samples

l = (p + 1)/6*p

b() = calls to the bernpoly module

Once calculated, the coefficients are stored in the global array Ar for module Queue to use.

The parameters inputed to the ArCoef module are r, p, and l. r plus one is the number of coefficients to be evaluated, always starting with zero. A[0] is always set to 1.0. p and l are described above.

There are no parameters outputed from ArCoef. The global array Ar is updated.

bernpoly is the only module which ArCoef needs to call. The values returned by the function bernpoly are used to calculate the Ar coefficients as shown in equation (16) above.

The temporary internal variables used are sign, partial, ptor, k, rt, and pl. sign is set to either plus or minus one as determined by raising minus one to the kth power, and represents the final sign of the kth coefficient being calculated. partial is temporary holding variable for intermediate results. The integer k ranges from zero to r, and repesents the current coefficient being evaluated. rt and pl are variables which are passed to the module bernpoly and contain the values r plus one and p times l respecfully.

The first step in ArCoef is to set the first element in the array Ar (Ar[0]) to one. To evaluate the next r terms the temporary variables sign, ptor, rt and pl are evaluated, bernpoly is called, and then each coefficient is calculated by equation (16).

ArCoef is an efficient procedure to calculate Ar coefficients required by the procedure Queue. The resulting coefficients are stored in the global array Ar.

Queue

The Queue module is a procedure which calculates the Q coefficients that are required by the modR module. The Q coefficients are calculated by the equation

$$Q_r = \frac{1}{r} \sum_{k=1}^{r} k A_k Q_{r-k}, \quad Q_0 = 1 \tag{17}$$

where

r = the term being evaluated

k = the kth coefficient

A = the Ar coefficients

Q = the Q coefficients

Once calculated, the coefficients are stored in the global array Q for module modR to use.

The paramters inputed into the Queue module are r, p, and l. r plus one is the number of coefficients to be evaluated, always starting with zero. Q[0] is always set to 1.0. p and l are passed to the module ArCoef.

There ae no parameters outputed from Queue. The global array Q is updated.

The module ArCoef is the only module which Queue needs to call. The values calculated by module ArCoef are stored in global array Ar and used by module Queue as shown in equation (17).

The temporay internal variables used by module Queue are i, k, and sum. The integer i ranges from zero to r and represents the current coefficient being evaluated. The integer k, ranging from one to i, is used as an internal loop counter for summing the terms that make up each Q coefficient. Real variable sum adds the intermediate terms together as they are being calculated in the internal loop.

The first step of module Queue is to call module ArCoef to calculate the Ar coefficients needed. The next step is to initialize the first Q coefficient Q[0] to 1.0. Finally, each q coefficient is calculated as shown in equation (17) starting with Q[1] and continuing until Q[r].

The module Queue is an efficient procedure to calculate the Q coefficients required by module modR. The source code for module Queue is straight forward translation of equation (17). The resulting Q coefficients are stored in global array Q.

CreateAjr

The CreateAjr module is a procedure which calculates the Ajr coefficients that are used by the CreateCjr module. The Ajr coefficients are calculated by the equation

$$A_{jr} = \frac{(-1)^{r-1}}{r(r+1)} \left[b_{r+1}(a) - b_{r+1}(a+v+j) \right], \; A_{j0} = 1 \qquad (18)$$

where

j and r = the row and column repectivly of the coefficient being evaluated

a and v = values passed to CreateAjr from CreateCjr

b() = calls to the bernpoly module.

Once calculated, the coefficients are stored in the global array Ajr for module CreateCjr to use.

The parameters inputed to the CreateAjr module are j, r, a, and v. Integers j and r define the size of the matrix of coefficients to be calculated for module CreateCjr. The real variables a and v are used as shown in equation (18) above.

There are no parameters outputed from module CreateAjr. The global array Ajr is updated.

bernpoly is the only module which CreateAjr needs to call. The values returned by the function bernpoly are used to calculate the Ajr coefficients as shown in equation (18) above.

The temporary internal variables needed by module CreateAjr are l, m, sign, rt, and temp. The integers l and m are used as loop counters to specify the current coefficient being evaluated. Integers sign and rt are used to speed up the execution time and make the source code more readable. sign alternates between plus and minus one and represents minus one raised to the r minus one power in equation (18). rt always contains the value of the column number, of the coefficient being evaluated, plus one. The real variable temp holds intermediate results which can be used by all the coefficients in the same column, this avoids recalculation of intermediate results which in turn speeds up the execution time of the module.

The first step is to initialize sign to positive one. All the values in column zero in array Ajr are also set to positive one. Once the variables are initialized the rest of the coefficients are evaluated, one at a time, using equation (18).

CreateAjr is an efficient procedure to calculate the Ajr coefficients required by procedure CreateCjr. The resulting coefficients are stored in the global array Ajr.

CreateCjr

The CreateCjr module is a procedure which calculates the Cjr coefficients that are used by module modR. The Cjr coefficients are calculated by the equation

$$C_{jr} = \frac{1}{r} \sum_{k=1}^{r} k A_{jk} C_{j,r-k} \quad , \quad C_{j,0} = 1 \qquad (19)$$

where

j and r = the row and column respectivly of the coefficient being evaluated

A = Ajr coefficients

C = Cjr coefficients.

Once calculated, the coefficients are stored in the global array Cjr for module modR to use.

The parameters inputed to the CreateCjr are j, r, a, and v. Integers j and r define the size of the matrix of coefficients to be calculated. The real variables a and v are passed to module CreateAjr and are not used directly by module CreateCjr.

There are no parameters outputed from module CreateCjr. The global array Cjr is updated.

CreateAjr is the only module which CreateCjr needs to call. The values created by module CreateAjr are used by module CreateCjr as shown in equation (19) as shown above.

The temporary internal variables used by CreateCjr are l, ..., k, and sum. The integers l, m, and k are all used as loop counters. l and m represent the column and row number of the current coefficient being evaluated and k is used, as shown in equation (19) above, to loop through the intermediate results which must be summed. The real variable sum holds the summation of the intermediate results for each coefficient.

When the module CreateCjr is first entered the module CreateAjr is called to calculate all the required Ajr coefficients. The next step is to initialize all the coefficients in column zero to positive one. The rest of the module calculates the remaining coefficients as described by equation (19) one at a time. The coefficients are calculated in row major order starting with zero and working up to column j, for columns one through r.

Module CreateCjr is an efficient implementation of equation (19). The calculated are stored in global array Cjr for module [illegible]R to use.

modR

The modR module is a procedure which calculates the R coefficients for the main module thesis, but which are used by the probability function prob. The R coefficients are calculated by the equation

$$\sum_{j=0}^{r} \left[R_{r-j} * C_{r-j,j} \right] = Q_r \qquad (20)$$

where

r = the rth term

Q = the Qr coefficients

C = the Cjr coefficients

Once calculated the R coefficients are stored in the global array R for module prob to use.

The parameters inputed into the modR module are j, r, a, v, p, and l. The only parameter module modR actually uses is r, which is the number of terms plus one it must calculate. The coefficients always start with R[0] and are calculated consecutively up to R[r]. R[0] is initialized to 1.0. The rest of the parameters are passed to modules Queue and CreateCjr to calculate the Q and Cjr coefficients required by modR.

There are no parameters outputed from module modR. The global array R is updated.

modR needs to call modules Queue and CreateCjr to calculate the Q and Cjr coefficients respectively. The coefficients are used as described in equation (20) above.

The temporary internal variables used by modR are i, k, and sum. The integers i and k are used as loop counters. The outside loop is controlled by i, which also represents the current coefficient being calculated. The inside loop counter is k. k ranges from one to i and represents the intermediate terms in the summation part of equation (20). The real variable sum holds the sum of the intermediate terms for each coefficient.

The first steps of module modR is to call modules Queue and CreateCjr. The rest of the module calculates the R coefficients one at a time by executing the following steps. First, evaluate the inside summation of equation (20). Then subtract the sum from the Q coefficient. Finally, divide the result by the Cjr coefficient. The final result is stored in the global array R and the next r coefficient can then be calculated.

Module modR is a straight forward and efficient implementation of equation (20).

loggam

The module loggam is a function which calculates the natural logrithm of gamma of a specific point sent to it by either the module prob or beta which call it. The equation used is shown below.

$$Loggam = (dy - 0.5) * ln(dy) + dp - dy - ln(dterm) + ((((((dz * ds + dw) * ds + dv) * ds + du) * ds + dt) * ds + dR) * ds + dq) / dy \tag{21}$$

where

dp, dz, dw, dv, du, dt, dr, and dq are constants

dy and dterm = variations of dx

ds = de / (dy * dy)

Once calculated the result is returned to the calling module.

The only parameter inputed into the function loggam is dx. The real variable dx is the point for which the natural logrithm of gamma is needed.

There are no parameters outputed. The only value returned is the result of the function.

The function loggam is a self-contained function and does not need to call any other modules to complete it's calculations.

All the temporary internl variables, except dy, dterm, and ds, are used as constants. The constants are used for clarity because the constants contain many digits and would make the finial equation unreadable. The variables dy, dterm, and ds are used as shown in equation (21).

The logic for the module loggam is straight forward. Once the value is determined to be within a meaningful range equation (21) is implemented. After the calculation is complete the result is returned to the calling module.

Function loggam is a fairly efficient implementation of the gamma function. Some efficiency is lost in assigning constants to variables, but the small amount lost was determined necessary to increase the readability. The natural logrithm of the result is returned because it is much easier to calculate.

beta

The function beta calculates the beta distribution result given the two points from ibeta by using the following equation

$$beta = \exp(\mathrm{loggam}(p) + \mathrm{loggam}(q) - \mathrm{loggam}(p+q)) \qquad (22)$$

where

loggam() = calls to the function loggam

p, q = points for which the function is evaluated

Once calculated the result is returned to ibeta.

The only parameters inputed into beta are p and q. The real variables p and q determine the point of the beta distribution required by the module ibeta.

There are no parameters outputed. The only value returned is the result of the function.

The logic of the function beta is contained in equation (22). Once called by ibeta, beta calls the function loggam three times, calculates the result as shown in equation (22), and returns the result to ibeta.

ibeta

The function ibeta calculates the incomplete beta function for modules prob and betap. The incomplete beta function, along with the R coefficients obtained by procedure modR, is the main function of the equation this thesis project is automated. The equation (11) can be found in the first chapter. The main equation use by module ibeta is shown below.

$$ibeta = exp(p * ln(x)) * exp((q-1) * ln(1-x)) / (beta(p,q) * p) * bint \qquad (23)$$

where

p, q, and x are input variables

beta() = a call to module beta

bint = error constant

The three input parameters p, q, and x are all real variables. There are no output variables. The only value the function ibeta returns to the calling modules prob and betap is the result of the function.

The internal variable bint is necessary because the calculation of the incomplete beta funcion is not a fix ratio between the input variables.

prob

The function prob calculates the theoretical probability for a given point x for system two. The main module thesis and module newton both need to call prob. The equation which represents module prob is shown below.

$$PROB = \left[\sum_{k=1}^{p-1} exp\left(loggam(n+k/p) - loggam(n)\right)\right] * \left[\sum_{i=0}^{Limit} R[i] * exp\left(loggam(n-1+a) - loggam(n-1+a+v+i)\right) * ibeta(n-1+a, v+i, x)\right] \tag{24}$$

where

ibeta() and loggam() are calls $\overset{to}{}$ modules

R[i] = R coefficients

a,v,p,l,x,n, and limit are input variables

The input variables a, v, p, l, x, and n are real and the variable limit is an integer. There are no output variables, only the result of the function is returned.

The internal integer variables i and k are used as loop counters for the summations required by equation (24). The other internal variables, reals tsum and t2sum, are used as temporary holding locations for the intermediate sums.

Function prob's logic is depicted by equation (24).

betap

The function betap calculates the inverse beta function. Function newton uses the result from function betap as an initial starting point for it's calculations. For further information about function newton look at the next section.

The parameters alpha, p and q are inputed into the module betap. With these starting parameters the module betap calculates the inverse beta function and returns the value to the calling module newton. No parameters are returned from betap, only the value of the function is outputed.

The only module betap needs to call is the module ibeta.

There are many internal variables but most are used as loop counters or constants. The integers i and j are used as loop counters. The real variables are used as constants.

newton

The function newton is called by the main module thesis to calculate the theoretical value system two was designed for. Newton's iterative method combined with the secant algorithm dictated the internal logic of the module newton.

The variables a, v, p, l, x, n and limit are all inputed into the module newton. All the variables are used as arguments to the probability function prob. There are no parameters outputed from module newton, but the result of the function is returned to the calling module thesis. Besides the module prob, module newton also calls module betap to get the starting value of it's process.

The internal logic can be described by the following steps given the equation f(x) = 0 and the initial approximations P_0 and P_1 with $P_0 \neq P_1$.

Step 1 Set i = 2.

Step 2 Set $P_i = P_{i-1} - \dfrac{f(P_{i-1})(P_{i-1} - P_{i-2})}{f(P_{i-1}) - f(P_{i-2})}$ (25)

Step 3 Determine if the procedure should be continued. If so, goto Step 4. If not, goto Step5.

Step 4 Add 1 to i and goto Step 2.

Step 5 The procedure is complete.

A simple linear approach would not have worked because of the behavior of the probability function. The above steps were implemented in a straight forward method.

thesis

The main module thesis contains the driver, which supervises the entire system, and subsystem three, which makes the final analysis. The module thesis is invoked when the user starts the system. The five modules getreal, lmodule, modR, prob, and newton are called by module thesis and will be described in the following paragraphs when the internal logic of procedure thesis is discussed.

The first step, for procedure thesis, is ask the user for the number of samples, sample size, and the file name where the data points are stored if batch processing is used. The module getreal is used to interface between the user and module thesis.

The next step is to call procedure lmodule to calculate the l-value. This represents the result of system one.

Before module modR is called to calculate the R coefficients, a rough estimate of the number of terms that will be required by the module prob is made. The estimate is made because the time to calculate R coefficients increases rapidly with the number of terms needed.

After procedure modR is called, successive calls to the module prob is made. Each time the call is made the number of terms is increased. With x set at one the results from prob will quickly approach one. After the functin prob approaches one with a predetermined tolerance, the rest of the terms are insignificant. This step limits the number of terms from infinity in equation (11) to an amount that can be resonably calculated without much lost in accuracy.

Next the user must enter the alpha level to be tested for. The module getreal is again used as an interface between the user and the program. The rest of the steps are contained in a loop controlled by the value of alpha. When the user enters an alpha level of zero the loop and the program will end, otherwise the user can test as many alpha levels as desired.

The last module that is called is module newton. Function newton returns the theoretical result that system two was designed for. Now with the results from systems one and two, system three is executed.

In the final step, system three compares the results from one and two and reports to the user it's conclusion. If the result from system two is larger than from one the hypothesis that all the data comes from the same sample can be rejected, otherwise a definite conclusion can not be made.

IV. Testing and Verification

This chapter will discuss the methods used to test and verify the procedures written for the modules described in chapter three. First, the terms boundry value testing, path testing, and drivers will be described. Next, individual testing of some modules will be discussed. Finally, the original requirements from chapter one will be re-examined to come up with the test data required to show whether the system developed does what it was intended to do.

Testing Methods

Boundry value testing, as used in this thesis, is setting each variable to it's extreme value and any special cases and making sure the progam still works as expected. For example, a positive variable that has a highest value of thirty should be tested for zero, one, twenty-nine, thirty, and thirty-one.

Path testing, as used in this thesis, is setting the input variables to different values to ensure each line of computer code is executed at least once. This is important not only to verify each statement works correctly, but also so as to not waste computer space with lines of code that can never be executed because of bad logic. An example of path testing is when the program has an if test making sure the then and else part are both executed.

Drivers are modules that are essentiall empty, their purpose is to call lower modules in the structure charts for testing. Also found in drivers may be assignment statements for the variables being passed to the lower modules and print statements for printing the results of the subroutine calls. The results of the subroutine calls are then checked against the expected results to verify that the subroutine is wroking as expected.. The use of drivers allows the lower modules to be tested completely before the higher modules are written. This is an important concept because, for example, if a program has five levels of modules and a result returned from the highest level is wrong the error would probably be in the highest level if drivers were used in previous testing, otherwise the problem could be in any of the modules in lower levels.

While each module is tested for all the necessay inputs to fully test the module a test plan is also written. The test plan is a check list of all the inputs and appears in tabular form. Each item can then be checked systematically. A copy of the test plan used for this project appears in Appendix B. When developing a test plan it is important to remember that some numbers can test many cases and by picking the test numbers carefully the number of tests needed to be run can be greatly reduced. Next, the testing method of the individual modules will be looked at.

getreal

The main purpose for getreal is to read numbers into the program. The numbers can come from a data file or be entered in interactively. All the numbers selected for test cases shown in Appendix B were tested both from a file and interactively. The program only recognizes the first six characters of the file name, therefore a short file name and one that is more than six characters long should be tested. The module getreal should accept integers and reals. The reals may or may not have digits before the decimal points. The module should give a bad data message for any characters besides digits and decimal points. A number having two decimal points is also not accepted.

lmodule

The number of data points needed by module lmodule can be determined by multiplying the number of samples times the sample size. If there is not enough data points given, when running batch, the program will abend and if there are too many the program will ignore the additional data points. The data points inputed into the module lmodule were both real and integer. A zero was also included in the data points. The case where the sample size was larger than the number of samples and also the reverse case was tested.

com

The module com calculates number of possible combinations given two numbers. There two inputs n and i were set to various numbers to test all cases. There are three cases to test for. The first is when i equals zero. The next case is when n and i are equal. The last case covers the remaining combinations of numbers. When i equals zero or n the result is one.

b

Module b calculates bernoulli numbers. The numbers are constant and only depend on the previous numbers calculated, therefore only a few were picked out to test. Two of the numbers picked were zero and one to test the lower boundry. Numbers three, five, and nine were also picked because they equate to zero. The last two cases needed were a positive and negative result.

System Two

The rest of the testing of the modules was similiar to the ones explained above. The module modR and the modules modR needs for it's calculations are just implementations of equations. The test was accomplished by hand calculating some numbers and then checking the results. Appendix B shows some of the numbers tested. The results returned by the module newton were compared against the results obtained by Dr. Nagarsenker in 1980.

thesis

The module thesis, which represents the entire system, was tested for two main cases. The first is does it work for known data. The other does it meet all the requirements stated in the first chapter and expaned throughout the thesis effort. The requirements which were tested for are explained in the rest of this chapter.

Requirements

One the first requirement is that the system be interactive. The system should be able to handle the user entering the data points interactivly and from a previously created data file. An other feature, added for the user's convenience, is that the user can test the data for multiple alpha levels without waiting for the recalculation of coefficients that do not change. The last feature is that when a user enters a bad character when entering interactive data the user should be able to re-enter just that one point and not have to restart from the beginning.

The test cases illustrated in Appendix B for the module thesis was enter interactively and from a file to test all the requirements. Most of the system is a black box to the user which simplifies the testing of the requirements.

V. Conclusion

This chapter will first re-examine the overall requirements of this thesis and discuss whether or not they were met. Then possible problems, along with suggestions on how to resolve them, will be discussed. Finally, as a way of concluding, future expansion of this thesis will be looked at.

Requirements

The overall thesis was broken into three subsystems. The overall project was to create an interactive statistical package to examine failure rates assuming an exponential distribution. System one calculated the l-value. System two calculated the theoretical value. System three compared the results from systems one and two and reported it's conclusion back to the user.

Each of the subsystems worked correctly for most of the test cases. Some of the minor problems will be discussed later. Test data from some well known results only proved that the requirements were more than met.

A faculty member, which helped to test the user friendliness of the system, found it easy to use after taking a few minutes to read the user manual in Appendix C. The response time was found to be reasonable for interactive work. The ability to change the alpha level, without having to preform most of the time consumming calculations over, greatly added to the usefullness of the entire system.

Problems and Suggestions

There were two basic problems found; but, neither were prohibitive. The first was if the user decided to store the data points in a data file and the user misspelled the file name when responding to the program's prompt the program would abend with a file not found message. This was not found to be much of a problem because the file name was one of the first questions asked of the user, so not much processing time was lost. The solution is to re-execute the program.

The second problem is if the sample size isn't more than two greater than the number of samples the program will not work correctly. This is because Pascal only has single precision and when both number are close the round off error is too great. This isn't a problem if the user has sufficient data for each sample. The two possible solutions are to take a couple of samples at a time or the better solution is to recode the project in a language which allows double precision such as C.

Future Expansion

This thesis only handles a small, but useful, set of possible distributions. The same procedure can be used to automate many of the others. This thesis can act as a starting point for a large statical package while also being useful until the rest get completed.

Appendix A Source Code

```
*****************************************************************
*                                                               *
*    NAME: getreal                                              *
*    FUNCTION: reads in real numbers from a specified file      *
*    INPUTS: filename                                           *
*    OUTPUTS: num, flag                                         *
*    GLOBAL VARIABLES USED: none                                *
*    GLOBAL VARIABLES CHANGED: none                             *
*    GLOBAL TABLES USED: none                                   *
*    GLOBAL TABLES CHANGED: none                                *
*    MODULES CALLED: none                                       *
*    CALLING MODULES: thesis, lmodule                           *
*                                                               *
*****************************************************************

   procedure getreal(var filename:text;
                          var num : real; var flag : boolean);
   var ch : char;  i : real;  decimal :boolean;
     begin
       num := 0.0;  flag := false;  decimal := false;
       if eoln(filename)
         then readln(filename);
       read(filename,ch);
       while (not eoln(filename)) and (ch = ' ') do
         read(filename,ch);
       if (ch>= '0') and (ch <= '9')
         then num := ord(ch) - ord('0')
         else if ch = '.'
           then decimal := true
           else flag := true;
       while (not eoln(filename)) and (not flag) and (ch <> ' ')
             and (not decimal) do
         begin
           read(filename,ch);
           if (ch >= '0') and (ch <= '9')
             then num := 10 * num + (ord(ch)-ord('0'))
             else if ch = '.'
               then decimal := true
               else if ch <> ' '
                 then falg := true
       end;
       i := 10;
       while (not eoln)and(not flag)and(ch <> ' ')and(decimal) do
         begin  read(filename,ch);
           if (ch >= '0') and (ch <= '9')
             then num := num + (ord(ch)-ord('0'))/i
             else if ch <> ' '
               then flag := true;
           i := i * 10
        end;
     if flag
       then while not eoln(filename) do  read(filename,ch)
  end;
```

```
******************************************************************
*                                                                *
*    NAME: lmodule                                               *
*    FUNCTION: calculates l value                                *
*    INPUTS: filename, s, n                                      *
*    OUTPUTS: lmodule                                            *
*    GLOBAL VARIABLES USED: none                                 *
*    GLOBAL VARIABLES CHANGED: none                              *
*    GLOBAL TABLES USED: none                                    *
*    GLOBAL TABLES CHANGED: none                                 *
*    MODULES CALLED: getreal                                     *
*    CALLING MODULES: thesis                                     *
*                                                                *
******************************************************************

function lmodule(filename : string; s, n : real):real;
var sum, add, mult, i, j, k : real;
    flag : boolean;  num, count : integer;
begin
  flag := false;
  sum := 0.0;  add := 0.0;  mult := 1.0;
  i := 0.0;  j := 0.0;  k := 0.0;
  while (j < n) and (not flag) do
    begin
      while (i < s) and (not flag) do
        begin
          if filename = 'input'
            then begin
              num := trunc(i);
              write(num+1,' . ');
              getreal(input,k,flag);
              writeln
              end
            else
              getreal(infile,k,flag);
          sum := sum + k;
          if flag
            then begin
              writeln('bad data');
              if filename = 'input'
                then flag := false
                end
            else i := i + 1.0
          end;
      sum := sum / s;  add := add + sum;
      mult := mult * sum;  sum := 0.0;
      i := 0.0;  j := j + 1
      end;
  add := add / n;  j := add;
  num := trunc(n) - 1;
  for count := 1 to num do  add := add * j;
  lmodule := mult / add
  end;
```

```
*****************************************************************
*                                                               *
*    NAME: com                                                  *
*    FUNCTION: calculates number of possible combinations       *
*    INPUTS: n, i                                               *
*    OUTPUTS: com                                               *
*    GLOBAL VARIABLES USED: none                                *
*    GLOBAL VARIABLES CHANGED: none                             *
*    GLOBAL TABLES USED: none                                   *
*    GLOBAL TABLES CHANGED: none                                *
*    MODULES CALLED: none                                       *
*    CALLING MODULES: b, bernpoly                               *
*                                                               *
*****************************************************************

function com(n, i : integer): real;
  var j : integer;
      prod : real;
  begin
    j := n - i;
    prod := 1.0;
    while i > 0 do
      begin
        prod := prod * (n / i);
        n := n - 1;
        i := i - 1
      end;
    while j > 0 do
      begin
        prod := prod * (n / j);
        n := n - 1;
        j := j - 1
      end;
    com := prod
  end;
```

```
****************************************************************
*                                                              *
*    NAME: b                                                   *
*    FUNCTION: calculates bernoulli numbers                    *
*    INPUTS: n                                                 *
*    OUTPUTS: none                                             *
*    GLOBAL VARIABLES USED: none                               *
*    GLOBAL VARIABLES CHANGED: bflag                           *
*    GLOBAL TABLES USED: none                                  *
*    GLOBAL TABLES CHANGED: barray                             *
*    MODULES CALLED: com                                       *
*    CALLING MODULES: bernpoly                                 *
*                                                              *
****************************************************************

procedure b(n : integer);
  var sum : real;
      i, j : integer;
  begin
    bflag := true;
    barray[0] := 1.0;
    if (n < 1) or (n > 30)
      then n := 0;
    for j := 1 to n do
      begin
        sum := 0.0;
        for i := 0 to (j-1) do
          sum := sum + com(j+1,i) * barray[i];
        barray[j] := ((-1)/(j+1)) * sum
      end;
    for j := 1 to n do
      if (barray[j] < 0.0000001) and (barray[j] > -0.0000001)
        then barray[j] := 0.0
  end;
```

```
****************************************************************
*                                                              *
*    NAME: bernpoly                                            *
*    FUNCTION: evaluates bernoulli poloynomials                *
*    INPUTS: n, x                                              *
*    OUTPUTS: bernpoly                                         *
*    GLOBAL VARIABLES USED: bflag, num                         *
*    GLOBAL VARIABLES CHANGED: none                            *
*    GLOBAL TABLES USED: barray                                *
*    GLOBAL TABLES CHANGED: none                               *
*    MODULES CALLED: b, com                                    *
*    CALLING MODULES: Arcoef, createAjr                        *
*                                                              *
****************************************************************

function bernpoly(n : integer; x : real) : real;
  var k : integer;
      sum : real;
      power : real;
  begin
    sum := 0.0;
    power := 1.0;
    if (not bflag)
      then b(num);
    for k := n downto 0 do
      begin
        sum := sum + com(n,k) * barray[k] * power;
        power := power * x
      end;
    bernpoly := sum
  end;
```

```
*****************************************************************
*                                                               *
*    NAME: ArCoef                                               *
*    FUNCTION: calculates Ar coefficients for module Queue      *
*    INPUTS: r, p, l                                            *
*    OUTPUTS: none                                              *
*    GLOBAL VARIABLES USED: none                                *
*    GLOBAL VARIABLES CHANGED: none                             *
*    GLOBAL TABLES USED: none                                   *
*    GLOBAL TABLES CHANGED: Ar                                  *
*    MODULES CALLED: bernpoly                                   *
*    CALLING MODULES: Queue                                     *
*                                                               *
*****************************************************************

procedure ArCoef(r : integer; p,l : real);
  var sign : integer;
      partial : real;
      ptor : real;
      k : integer;
      pl : real;
  begin
    ptor := 1.0;
    pl := p * l;
    Ar[0] := 1.0;
    sign := 1;
    for k := 1 to r do
      begin
        ptor := ptor * p;
        sign := sign * -1;
        partial := (bernpoly(k+1,pl)-
                    p*ptor*bernpoly(k+1,l))/ptor;
        Ar[k] := (sign/(k*(k+1)))*partial
        end
  end;
```

```
****************************************************************
*                                                              *
*    NAME: Queue                                               *
*    FUNCTION: creates Q coefficients for module modR          *
*    INPUTS: r, p, l                                           *
*    OUTPUTS: none                                             *
*    GLOBAL VARIABLES USED: none                               *
*    GLOBAL VARIABLES CHANGED: none                            *
*    GLOBAL TABLES USED: Ar                                    *
*    GLOBAL TABLES CHANGED: Q                                  *
*    MODULES CALLED: ArCoef                                    *
*    CALLING MODULES: modR                                     *
*                                                              *
****************************************************************

procedure Queue(r : integer; p,l : real);
  var i,k : integer;
      sum : real;
  begin
    ArCoef(r,p,l);
    Q[0] := 1.0;
    for i := 1 to r do
      begin
        sum := 0.0;
        for k := 1 to i do
          sum := sum + k * Ar[k] * Q[i-k];
        Q[i] := (1/k) * sum
        end
  end;
```

```
*****************************************************************
*                                                               *
*    NAME: createAjr                                            *
*    FUNCTION: creates Ajr coefficients for createCjr           *
*    INPUTS: j, r, a, v                                         *
*    OUTPUTS: none                                              *
*    GLOBAL VARIABLES USED: none                                *
*    GLOBAL VARIABLES CHANGED: none                             *
*    GLOBAL TABLES USED: none                                   *
*    GLOBAL TABLES CHANGED: Ajr                                 *
*    MODULES CALLED: bernpoly                                   *
*    CALLING MODULES: createCjr                                 *
*                                                               *
*****************************************************************

procedure createAjr(j,r : integer; a,v : real);
  var l,m : integer;
      sign, rt : integer;
      temp : real;
  begin
    sign := 1;
    for m := 0 to j do
      Ajr[m,0] := 1.0;
    for l := 1 to r do
      begin
        rt := l + 1;
        temp := bernpoly(rt,a);
        for m := 0 to j do
          Ajr[m,l] := (sign/((l)*(l+1)))*
                      (temp-bernpoly(rt,a+v+m));
        sign := sign * (-1)
        end
  end;
```

```
*****************************************************************
*                                                               *
*    NAME: createCjr                                            *
*    FUNCTION: creates Cjr coefficients for modR                *
*    INPUTS: j, r, a, v                                         *
*    OUTPUTS: none                                              *
*    GLOBAL VARIABLES USED: none                                *
*    GLOBAL VARIABLES CHANGED: none                             *
*    GLOBAL TABLES USED: Ajr                                    *
*    GLOBAL TABLES CHANGED: Cjr                                 *
*    MODULES CALLED: createAjr                                  *
*    CALLING MODULES: modR                                      *
*                                                               *
*****************************************************************

procedure createCjr(j,r : integer; a,v : real);
  var l,m : integer;
      k : integer;
      sum : real;
  begin
    createAjr(j,r,a,v);
    for m := 0 to j do
      Cjr[m,0] := 1.0;
    for l := 1 to r do
      for m := 0 to j do
        begin
          sum := 0.0;
          for k := 1 to l do
            sum := sum + k * Ajr[m,k] * Cjr[m,l-k];
          Cjr[m,l] := sum / l
        end
  end;
```

```
*****************************************************************
*                                                               *
*    NAME: modR                                                 *
*    FUNCTION: creates r coefficients for function prob         *
*    INPUTS: j, r, a, v, p, l                                   *
*    OUTPUTS: none                                              *
*    GLOBAL VARIABLES USED: none                                *
*    GLOBAL VARIABLES CHANGED: none                             *
*    GLOBAL TABLES USED: Q, Cjr                                 *
*    GLOBAL TABLES CHANGED: R                                   *
*    MODULES CALLED: Queue, createCjr                           *
*    CALLING MODULES: thesis                                    *
*                                                               *
*****************************************************************

procedure modR(j,r : integer; a,v,p,l : real);
  var sum : real;
      i,k : integer;
  begin
    createCjr(j,r,a,v);
    Queue(r,p,l);
    R[0] := 1.0;
    for i := 1 to r do
      begin
        sum := 0.0;
        for k := 1 to i do
          sum := sum + r[i-k] * Cjr[i-k,k];
        R[i] := (Q[i] - sum) / Cjr[i,0]
        end
  end;
```

```
*****************************************************************
*                                                               *
*    NAME: loggam                                               *
*    FUNCTION: calculates the logrithm of gamma of n            *
*    INPUTS: dx                                                 *
*    OUTPUTS: loggam                                            *
*    GLOBAL VARIABLES USED: none                                *
*    GLOBAL VARIABLES CHANGED: none                             *
*    GLOBAL TABLES USED: none                                   *
*    GLOBAL TABLES CHANGED: none                                *
*    MODULES CALLED: none                                       *
*    CALLING MODULES: beta, prob                                *
*                                                               *
*****************************************************************

function loggam(dx : real):real;
  var rdo,dy,dterm,de,da,db,domeg,dlggm : real;
      ds,dz,dw,dv,du,dt,dr,dq,dp : real;
  begin
    rdo := 0.0;  dy := dx;
    dterm := 1.0;  de := 1.0;
    domeg := 1.0e25;
    da := 0.999999999;  db := 1.000000001;
    dlggm := domeg;
    if (dx >= rdo)
      then begin
        dlggm := rdo;
        if ((dx<=da)or(dx>=db))
          then begin
            if ((dx<=(da+de))or(dx>=db+de)))
              then begin
                while ((dy-18.00)<=0.0) do
                  begin
                    dterm := dterm * dy;
                    dy := dy + de
                    end;
                ds := de / (dy * dy);
                dz := 0.005410256410256410;
                dw := -0.001917526917526918;
                dv := 0.000841750841750841 8;
                du := -0.000595238095238095 2;
                dt := 0.000793650793650793 7;
                dr := -0.002777777777777778;
                dq := 0.083333333333333333;
                dp := 0.91893853320467274;
                dlggm := (dy-0.5)* ln(dy)+dp-dy-ln(dterm);
                dlggm := dlggm + ((((((dz*ds+dw)*ds+dv)*ds+du)*
                              ds+dt)*ds+dr)*ds+dq)/dy
                end
          end
    end;
  loggam := dlggm
  end;
```

```
*****************************************************************
*                                                               *
*    NAME: beta                                                 *
*    FUNCTION: calculates the beta of p and q                   *
*    INPUTS: p, q                                               *
*    OUTPUTS: beta                                              *
*    GLOBAL VARIABLES USED: none                                *
*    GLOBAL VARIABLES CHANGED: none                             *
*    GLOBAL TABLES USED: none                                   *
*    GLOBAL TABLES CHANGED: none                                *
*    MODULES CALLED: loggam                                     *
*    CALLING MODULES: ibeta                                     *
*                                                               *
*****************************************************************

function beta(p,q : real):real;
  begin
    beta := exp(loggam(p)+loggam(q)-loggam(p+q))
    end;
```

```
*****************************************************************
*                                                               *
*    NAME: ibeta                                                *
*    FUNCTION: calculates the incomplete beta function          *
*    INPUTS: p, q, x                                            *
*    OUTPUTS: ibeta                                             *
*    GLOBAL VARIABLES USED: none                                *
*    GLOBAL VARIABLES CHANGED: none                             *
*    GLOBAL TABLES USED: none                                   *
*    GLOBAL TABLES CHANGED: none                                *
*    MODULES CALLED: beta                                       *
*    CALLING MODULES: prob, betap                               *
*                                                               *
*****************************************************************

function ibeta(p,q,x : real):real;
var bint,pp,qq,dp,xx,rxx,t,d,b,s,temp1,temp2 : real;
    ick,iq,ip,iqm,i : integer;
begin
  bint := 0.0;
  if ((x=0.0)or(x=1.0)) then bint := x
    else begin
      pp := p;
      qq := q;
      ick := 0;
      xx := x;
      iq := trunc(q);
      dq := iq;
      ip := trunc(p);
      dp := ip;
      if ((dq=q)or(dp=p))
        then begin
          if (dp = p)
            then begin
              iq := ip;
              pp := q;
              qq := p;
              ick := 1;
              xx := 1.0 - x
              end;
        t := 1.0;
        rxx := xx/(1.0-xx);
        bint := t;
        iqm := iq - 1;
        if (iqm <> 0)
          then for i := 1 to iqm do
            begin
              d := i;
              t := t*rxx*(qq-d)/(pp+d);
              bint := bint + t
              end
      end
    else begin
      if (xx>(pp/(pp+qq)))
        then begin
          pp := q;
          qq := p;
          xx := 1.0 - x;
          ick := 1
          end;
      bint := 1.0;
      t := 1.0;
      iq := trunc(qq+(1.0-xx)*(pp+qq));
      s := iq;
      if (iq = 0)
        then begin
          t := 1.0 - xx;
          bint := t
          end
        else begin
          if (iq<>1)
            then begin
              rxx := xx/(1.0-xx);
              iqm := iq - 1;
              for i := 1 to iqm do
                begin
                  d := i;
                  t := t * rxx * (qq-d)/(pp+d);
                  bint := bint + t
                  end
            end;
          t := t * xx * (qq - s)/(pp+s);
          bint := bint + t
      end;
      i := 1;
      while((i<101)and
           (abs(t/bint) (0.00000000000000000000000001))) do
        begin
          d := i;
          t := t * (pp+qq+d-1.0)/(pp+s+d)*xx;
          bint := bint + t;
          i := i + 1
          end
    end;
    b := beta(p,q);
    temp1 := exp(pp*ln(xx));
    temp2 := exp((qq-1.0)*ln(1.0-xx));
    bint := temp1*temp2/(b*pp)*bint;
    if (ick = 1)
      then bint := 1.0 - bint
    end;
  ibeta := bint
end;
```

```
*****************************************************************
*                                                               *
*    NAME: prob                                                 *
*    FUNCTION: calculates the distribution function of l        *
*    INPUTS: a,v,p,l,x,n,limit                                  *
*    OUTPUTS: prob                                              *
*    GLOBAL VARIABLES USED: none                                *
*    GLOBAL VARIABLES CHANGED: none                             *
*    GLOBAL TABLES USED: R                                      *
*    GLOBAL TABLES CHANGED: none                                *
*    MODULES CALLED: loggam, ibeta                              *
*    CALLING MODULES: thesis, newton                            *
*                                                               *
*****************************************************************

function prob(a,v,p,l,x,n:real;limit:integer):real;
  var i,j,k : integer;
      tsum,t2sum : real;
  begin
    t2sum := 0.0;
    tsum := 1.0;
    j := trunc(p-1);
    for k := 1 to j do
      tsum := tsum * exp(loggam(n+k/p)-loggam(n));
    for i := 0 to limit do
      t2sum := t2sum + R[i]*exp(loggam(n-1+a)-loggam(n-1+a+v+i))*
                       ibeta(n-1+a,v+i,x);
    prob := tsum * t2sum
  end;
```

```
*****************************************************************
*                                                               *
*    NAME: betap                                                *
*    FUNCTION: calculates the inverse beta function             *
*    INPUTS: alpha,p,q                                          *
*    OUTPUTS: betap                                             *
*    GLOBAL VARIABLES USED: none                                *
*    GLOBAL VARIABLES CHANGED: none                             *
*    GLOBAL TABLES USED: none                                   *
*    GLOBAL TABLES CHANGED: none                                *
*    MODULES CALLED: ibeta                                      *
*    CALLING MODULES: newton                                    *
*                                                               *
*****************************************************************

function betap(alpha,p,q : real):real;
var jj,jend,j,i : integer;
    dp,dl,dif,dlx,dux,dmp,decr,dmpu : real;
    dm,dn,du,dfd,dabf,dfune : real;
    flag : boolean;
    esp1,esp2,esp3,esp4 : real;
    darg,dfun : array[1..4] of real;
begin
  esp1 := 1.0e-180;  esp2 := 1.0e-13;
  esp3 := 1.0e-11;  esp4 := 1.0e-10;
  dp := alpha;  dm := p;
  dn := q;  du := 1.0;
  flag := true;
  if (((dp*(du-dp))<0)or(dm<0)or(dn<0))
    then dmp := 0.0
    else if ((dp*(du-dp))=0)
      then dmp := alpha
      else if (dm = 1.0)
        then dmp := du-exp((du/dn)*ln(du-dp))
        else if (dn = 1.0)
          then dmp := exp(du/dm)*ln(dp))
          else flag := false;
  if (not flag)
    then begin
      dl := 0.0;  dif := 1.0/3.0;
      dlx := -dp;  dux := du-dp;
      jj := 0;
      dmpu := 0.0;
      jend := 3;
      while ((jj<25) and (not flag)) do
        begin
          if jj = 25 then jend := 3;
          jj := jj + 1;
          j := 1;
          while ((j<=jend) and (not flag)) do
            begin
              dmp := (du+dl)/2.0;
              i := 1;
  if ((du-dl)<espl)
    then flag := true
    else if (((du-dl)<(esp2*dp))and(dl>esp2))
      then flag := true
      else while ((i<3) and (not flag)) do
        begin
          darg[i] := dl+(du-dl)*dif*i;
          dfun[i] := ibeta(dm,dn,drg[i])-dp;
            if (dfun[i]=0) then dmp := darg[i];
            if (dfun[i]=0) then flag := true
              else if ((dfun[i]<0.0)and(i=2))
                then begin
                  dl := darg[2];  dlx := dfun[2]
                  end
                else if (dfun[i]>0.0)
                  then begin
                    du := darg[i];  dux := dfun[i];
                    if (i = 2)
                      then begin
                        dl:=darg[1]; dlx := dfun[1]
                        end
                      else i := 2
                    end;
                    i := i + 1
          end;
          j := j + 1
end;
if (not flag)
  then begin
    jend := 2;  dmp := (du+dl)/2.0;
    dfd := dux-dlx;
    if ((dfd<esp3)and(dfd<esp4*dp)))
      then flag := true
    end;
if (not flag)
  then begin
    decr := dux * (du-dl)/dfd;  dmp := du - decr;
    if (((dmp-dl)<espl)or(((dmp-dl(esp2)
          and(dl>esp2))) then flag := true
    end;
if (not flag)
  then begin
    dfun[3] := ibeta(dm,dn,dmp)-dp;
    dabf := abs(dfun[3]);  dfune := dfun[3];
    if (((dabf<esp3)and(dabf<(esp4*dp)))or
       (dmp<espl)or(((du-dmpu)<esp2)and
       (du>0.999999999999))or(dfun[3]=0.0))
       then flag := true
  end;
if (not flag)
  then begin
    if (dfun[3]<0.0)
      then begin
```

```
         if (decr<(0.9*(du-dl)))
          then begin
           dl := dmp;  dlx := dfune
           end
          else begin
           dmpu := dmp;  dmp := 5.0*(dmp-dl)+dl;
           dfune := ibeta(dm,dn,dmp)-dp;
           if (dfune = 0.0)
            then flag := true;
           if (not flag)
            then begin
             if (dfune < 0.0)
              then begin
               dl := dmp;  dlx := dfune
               end
              else begin
               du := dmp;  dux := dfune;
               dl := dmpu;  dlx := dfun[3]
               end
             end
           end
         end
        else begin
         if (decr>=(0.1*(du-dl)))
          then begin
           du := dmp;  dux := dfune
           end
          else begin
           dmpu := dmp;
           dmp := du-5.0*decr;
           dfune := ibeta(dm,dn,dmp)-dp;
           if (dfune=0.0) then flag := true;
           if (not flag)
            then begin
             if (dfune<0.0)
              then begin
               du := dmpu;
               dux := dfun[3];
               dl := dmp;
               dlx := dfune
               end
              else begin
               du := dmp;
               dux := dfune
               end
             end
           end
         end
       end
     end
   end;
 betap := dmp
end;
```

```
**************************************************************
*                                                            *
*    NAME: newton                                            *
*    FUNCTION: calculates the percentage point using         *
*              newton's approximation                        *
*    INPUTS: a,v,p,l,alpha,n,limit                           *
*    OUTPUTS: newton                                         *
*    GLOBAL VARIABLES USED: none                             *
*    GLOBAL VARIABLES CHANGED: none                          *
*    GLOBAL TABLES USED: none                                *
*    GLOBAL TABLES CHANGED: none                             *
*    MODULES CALLED: betap, prob                             *
*    CALLING MODULES: thesis                                 *
*                                                            *
**************************************************************

function newton(a,v,p,l,alpha,n : real; limit : integer):real;
  var z,pa : array[1..25] of real;
      sp : real;
      k : integer;
      done : boolean;
  begin
    z[1] := betap(alpha,(n-(p+1)/(6*p)),v);
    if p < 3
      then z[2] := z[1] + 0.05
      else z[2] := z[1] - 0.05
    x := z[1];
    sp := prob(a,v,p,l,x,n,limit);
    pa[1] := sp;
    k := 2;
    done := false;
    while((k<25)and(sp<=1.0)and(sp>=0.0)and(not done)) do
      begin
        x := z[k];
        sp := prob(a,v,p,l,x,n,limit);
        pa[k] := sp;
        if (abs(sp-alpha) 0.0000001)
          then done := true
          else z[k+1] := z[k]-(z[k]-z[k-1])*
                 (pa[k]-alpha)/(pa[k]-pa[k-1]);
        k := k + 1
        end;
    if ((sp>1.0) or (sp<(0.0))
      then writeln('incorrect prob value',x,sp);
    newton := z[k-1]
    end;
```

```
*****************************************************************
*                                                               *
*    NAME: thesis                                               *
*    FUNCTION: manages overall system                           *
*    INPUTS: none                                               *
*    OUTPUTS: none                                              *
*    GLOBAL VARIABLES USED: none                                *
*    GLOBAL VARIABLES CHANGED: bflag                            *
*    GLOBAL TABLES USED: none                                   *
*    GLOBAL TABLES CHANGED: none                                *
*    MODULES CALLED: getreal, lmodule, modR, prob, newton       *
*    CALLING MODULES: none                                      *
*                                                               *
*****************************************************************

program thesis (input,output);
  type string = packed array[1..5] of char;
       numary = array[0..30] of real;
       numary2 = array[0..30,0..30] of real;
  var filename : string;
      bflag,flag : boolean;
      l,n,t,x,a,v,p,prb,tsum,alpha,s : real;
      limit,i,j,k,r,num : integer;  barray,R,Ar,Q : numary;
      Ajr,Cjr : numary2;  infile : text;
  begin
    flag := false;  bflag := false;
    writeln('enter number of samples');
    getreal(input,p,flag);
    while (flag) do
      begin
        writeln('error enter number');
        getreal(input,p,flag)
        end;
    writeln('enter sample size');
    getreal(input,n,flag);
    while (flag) do
      begin
        writeln('error enter number');
        getreal(input,n,flag)
        end;
    writeln('enter tthe file name where the data is stored');
    writeln('or "input" for entering the data interactivly');
    i := 1;  filename[5] := ' ';  readln;
    while (i<6) and (not eoln) do
      begin
        read(filename[i]);
        i := i + 1
        end;
    if filename <> 'input'
      then reset(infile,filename);
    l := lmodule(filename,n,p);
```

```
if (n >= 20)
  then num := 10
  else if ((n-p)>3.0)
         then num := 20
         else num := 22;
j := num;
r := num;
t := (p+1)/6*p);
v := (p-1)/2;
a := (1-v)/2;
modR(j,r,a,v,p,t);
x := 1.0;
prob := 0.0;
limit := 0;
while ((abs(prb-1.0)>0.0000001)and(limit<num)) do
  begin
    limit := limit + 1;
    prb := prob(a,v,p,t,x,n,limit)
    end;
alpha := 1.0;
if (limit >= num)
  then writeln('too few terms, change number of terms')
  else begin
    writeln('enter alpha level to be tested, 0 to end');
    getreal(input,alpha,flag);
    while flag do
      begin
        writeln('enter alpha level to be tested, 0 to end');
        getreal(input,alpha,flag)
        end
    end;
while (alpha <> 0) do
  begin
    prb := newton(a,v,p,t,alpha,n,limit);
    if (prb > 1)
      then writeln('reject hypothesis that all data')
      else writeln('cann't reject hypothesis that all data');
    writeln('  comes from the same sample');
    writeln('enter alpha level to be tested, 0 to end');
    getreal(input,alpha,flag);
    while flag do
      begin
        writeln('enter alpha level to be tested, 0 to end');
        getreal(input,alpha,flag)
        end
    end
  end
end.
```

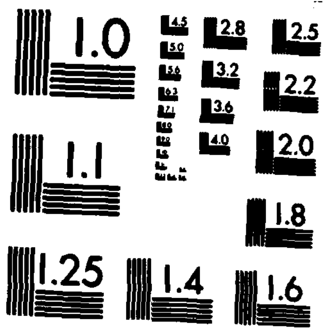
AD-A153 378 SOFTWARE ENVIRONMENT FOR CONDUCTING INTERACTIVE STATISTICAL ANALYSIS(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING 2/2

UNCLASSIFIED M F AMELL 14 DEC 84 AFIT/GCS/MATH/84D-2 F/G 12/1 NL

END

FILMED

DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1

Appendix B
Test Plan

Module	Input Data	Result
getreal	5 712 .3 0.3 10.2	accepted
	a5 71b 6c3 -5 2..1	bad data
lmodule	1 3 2 2	
	5 7 3 1	0.88889
	2 4 3	
	2 3 2	
	1 8 0	
	1.5 3.5 4	1.0
com	n = 5 i = 0	1.0
	5 5	1.0
	8 3	56.0
	9 7	36.0
	9 2	36.0
b	0	1.0
	1	-0.5
	3	0.0
	4	-0.03333
	5	0.0
	9	0.0
	10	0.07575
bernpoly	n = 1 x = 1	0.05
	2 1	0.1666667
	2 2	2.1666667
	2 0.5	-0.083333
	3 0.25	0.046875
	4 0.125	-0.02137
	5 0.33333	-0.020576
ArCoef	r = 10 p = 2 l = 0.25	
	Ar[1]	1.734723e-18
	9	-1.057474e-19
Queue	r = 10 p = 2 l = .25	
	Q[2]	-1.562500e-02
	7	7.194460e-19
createAjr	j = r = 10 a = .25 v = .5	
	Ajr[0,1]	-1.734723e-19
	0,5	9.974660e-19

```
Module              Input Data                          Result
------              ----------                          ------

createCjr           j = r = 10  a = .25  v = .5
                    Cjr[5,1]                            -13.75
                        5,6                           15803.05

modR                j=r=10  a=l=.25  v=.5  p=2
                    R[2]                              2.602085e-18
                     9                               -6.394667e-15

loggam              dx = 21                           42.33561646
                         0.5                          0.572364943

beta                p = 20.75  q = .5                 0.391455
                        21       1.5                  9.0486e-03

ibeta               p=20.75  q=.5  x=.667             4.5376e-05
                      21      .5    .911              4.9243e-02

prob        a=.25 v=.5 p=2 l=.25 x=.911 n=21 limit=10    0.049
             .25   .5   2   .25   .912  21       10      0.050

betap                  alpha=.05  p=20.75  q=.5         0.910576

newton            same as prob  plus  alpha 0.05          .9116

thesis               sample 1                sample 2
                        6                       1
                        6                       1
                        6                       2
                        6                       2
                        7                       3
                        9                       4
                       10                       4
                       10                       5
                       11                       5
                       13                       8
                       16                       8
                       17                       8
                       19                       8
                       20                      11
                       22                      11
                       23                      12
                       25                      12
                       32                      15
                       32                      17
                       34                      22
                       35                      23

                    alpha = 0.05                        reject
                    alpha = 0.01                   cann't reject
```

Appendix C

User's Guide

This appendix will first show a simple example of how to run the thesis program. The computer prompts will then be all explained. Finally, the option for entering the data interactively will be further explained. In the following example the user responses are designated by a "+ ", although the prompt will not be seen by the user when running the thesis program.

```
enter number of samples
+ 3
enter sample size
+ 7
enter the file name where the data is stored
or "input" for entering the data interactively
+ data
enter alpha level to be tested, 0 to end
+ 0.05
reject hypothesis that all the data
  comes from the same population

enter alpha level to be tested, 0 to end
+ 0
```

The first two prompts ask for the number of samples, and the size of each sample to be tested. Each sample must have the same number of data points in it. The total number of data points the program expects can be found by multiplying the number of sample times the sample size.

The next prompt tells the program where to find the test data. If the user enters a name other than "input" the name is assumed to be the name of the file where all the data points are stored. When entering a file name there are three possible next steps for the program. First, if the file doesn't exist the program will abend will a file not found message. Next, if the file does exist but there are not enough data points in the file or non numeric symbols are found the program will end with a bad data message. The last, and the one looked for, alternative is that the program found the file and enough data points and then the program continues by giving the next prompt. The number of data points needed is discussed in the previous paragraph. If more data points are given then required the excess will be ignored. The response "input" will be examined in detail later in this appendix.

The program can abend with one other message which is "too few terms, change number of terms." The last message indicates to the user that a theoretical value can not be calculated for the number of samples and sample size numbers given because of the limitation of Pascal using single precision. The problem can be overcome by taking fewer samples or increasing the sample size. One attempt at fixing this problem is to convert the program to a language that has double precision such as C.

The last prompt before the results are determined is to enter the alpha level. The program will continue to prompt for different alpha levels to be tested until a zero (0) is entered ending the loop and the program.

The above example shows one of the two possible results of the thesis program's calculation which is "reject hypothesis that all the data comes from the same sample." The other possible result is "can't reject hypothesis that all the data comes from the same sample."

The response "input" to the file name prompt allows the user to enter the data interactively. The user will be prompted by a number followed by a period, which indicates the nth number within the particular sample. The prompt number runs from one to the sample size for each sample. If the user enters anything except a positive real number the program will prompt the same again indicating the character string was ignored.

Bibliography

1. Almes, Guy and Robertson, George "An Extensible File System for Hydra," Technical Report, Computer Science Department, Carnegie-Mellon University (1977).

2. Ambler, A.L. and others. "GYPSY: A Language for Specification and Implementation of Verifiable Programs," Proceedings of the ACM Conference on Language Design for Reliable Software, SIGPLAN Notices 12,3:1-10 (March 1977).

3. Dahl, Ole-Johan and others. Structured Programming. Academic Press, 1972.

4. Dale, Nell and Orshalick, David. Introduction to PASCAL and Structured Design. D.C. Health and Company, 1983.

5. Dijkstra, Edsger W. and others. Structured Programming. Academic Press, 1972.

6. Dijkstra, Edsger W. "The Structure of THE-Multiprogramming System," Communications of the ACM 11(5):341-346 (May 1968).

7. Department of Defense. Requirements for High Order Computer Programming Languages. SIGPLAN Notices 12(12):39-54, Dec 1977.

8. Flon, Lawrence. "Program Design With Abstract Data Types," Technical Report, Computer Science Department, Carnegie-Mellon University (1975).

9. Geschke, Charles and others. "Early Experience With Mesa," Communications of the ACM 20(8):540-552 (August 1977).

10. Habermann, A. Nico and others. "Modularization and Hierarchy in a Family of Operating Systems," Communications of the ACM 19(5):266-272 (May 1976).

11. Jensen, Kathleen and Wirth, Niklaus. Pascal User manual and Report. Springer-Verlag, 1974.

12. Liskov, Barbara H. "An Introduction to CLU," New Directions in Algorithmic Languages (1976).

13. Liskov, Barbara H. "The Design of the VENUS Operating System," Communications of the ACM 15(3):144-149 (March 1972).

14. Neumann, Peter G. and others. "On the Design of a Provably Secure Operating System," Proceedings of the IRIA Workshop on Protection in Operating Systems, Paris:161-175 (August 1974).

15. Norman, Donald A. "The Trouble With UNIX," Datamation (November 1981).

16. Organick, Eliot I. "The Multics System: An Examination of its Structure," MIT Press.

17. Parnas, David L. "On the Design and Development of Program Families," IEEE Transactions on Software Engineering 2(1):1-8 (March 1976).

18. Parnas, David L. "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM 15(12):1053-1058 (December 1972).

19. Parnas, David L. "Some Conclusions From an Experiment in Software Engineering Techniques," Proceedings Fall Joint Computer Conference, Vol 41:325-329 (1972).

20. Parnas, David L. "On a Buzzword: Hierarchical Structure," Proceedings of the IFIPS Congress 74 (1974).

21. Richie, D.M. and Thompson, K. "The UNIX Time-Sharing System," Communications of the ACM 17(7):365ff (July 1974).

22. Rochkind, Marc J. "The Source Code Control System," IEEE Transactions on Software Engineering 1(4):364-370 (Dec 1974).

23. Saxena, A.R. "A Verified Specification of a Hierarchical Operating System," Technical Report, Stanford University (1976).

24. Wirth, Niklaus. "Modula: A Language for Modular Programming," Software Practice and Experience 7(1):3-35 (1977).

25. Wulf, William A. and others. "An Introduction to the Construction and Verification of Alphard Programs," IEEE Transactions on Software Engineering 2(4):253-265 (Dec 1976).

VITA

Captain Mark F. Amell was born on 7 July 1957 in Buffalo, New York. He graduated from high school in East Hampton, Connecticut, in 1975 and attended the University of Hartford for one year. He then transferred to the University of Connecticut from which he received the degree of Bachelor of Science in Computer Engineering in January 1980. Upon graduation, he received a commission in the USAF through the ROTC program. He was called to active duty in March 1980. He served as a computer analysist for the Defense Communications Engineering Center, until entering the School of Engineering, Air Force Institute of Technology, in June 1983.

Permanent address: P.O. Box 120
East Hampton, Connecticut 06424

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: UNLIMITED
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): AFIT/GCS/MATH/84D-2	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: AFIT — 6b. OFFICE SYMBOL *(If applicable)*: ENC	7a. NAME OF MONITORING ORGANIZATION: AFIT/ENC
6c. ADDRESS *(City, State and ZIP Code)*: Wright-Patterson AFB Dayton, OH 45433	7b. ADDRESS *(City, State and ZIP Code)*: Wright-Patterson AFB Dayton, OH 45433
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: AFIT ENC — 8b. OFFICE SYMBOL *(If applicable)*: ENC	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State and ZIP Code)*: Wright-Patterson AFB Dayton, OH 45433	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT NO.
11. TITLE *(Include Security Classification)*: Software Environment for Conducting	
12. PERSONAL AUTHOR(S): Capt. Mark F. Amell	
13a. TYPE OF REPORT: Thesis — 13b. TIME COVERED FROM 1 sep 83 TO 14 dec 84	14. DATE OF REPORT *(Yr., Mo., Day)*: 84 Dec 14 — 15. PAGE COUNT: 103
16. SUPPLEMENTARY NOTATION	

Approved for public release: IAW AFR 190-17.
[illegible] 21 Feb 85
LYNN E. WOLAVER
Dean for Research and Professional Development
Air Force Institute of Technology (AFSC)
Wright-Patterson AFB OH 45433

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB. GR.	

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This study developed a software environment for conducting statistical analysis of exponential analysis of exponential failure models interactively. Exponential failure models will work for small amounts of data and are particularity valuable when gathering large amounts of data is not practical, such as loss of lives or airplanes.

The design of the system down to the individual modules deminstrated that using the software engineering techniques explored an environment can be efficiently created systematically. The approach used in this project not only created a valuable tool, but it's use is encouraged in further developments for other distributions.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: UNCLASSIFIED/UNLIMITED XX SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL: Dr. Panna Nagarsenker	22b. TELEPHONE NUMBER *(Include Area Code)*: (513) 255-7210	22c. OFFICE SYMBOL: ENC

DD FORM 1473, 83 APR — EDITION OF 1 JAN 73 IS OBSOLETE.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

11. Interactive Statistical Analysis